An ultra-quick primer to C++

Simon Scheidegger
simon.scheidegger@unil.ch
August 30th, 2018

Cowles Foundation – Yale

General information

Simon Scheidegger - simon.scheidegger@unil.ch for questions outside the class.

CPP Intros on the World Wide Web:

- http://www.learncpp.com/
- -...

CPP literature:

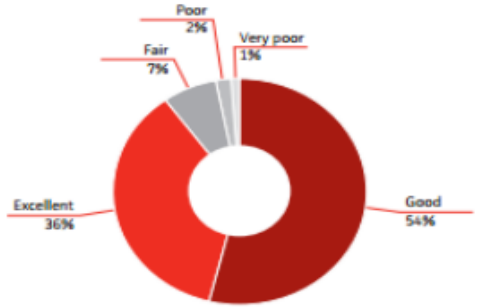
- “C++ Primer Plus (6th Edition)“ by Stephen Prata
- -...

CSCS* – Annual report 2016 – Sad, so sad

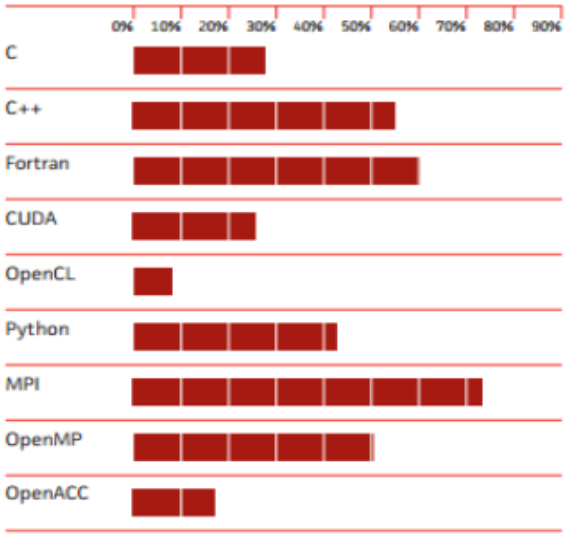
Application Development

Do you develop and maintain application codes?

How do you rate the offered range of programming tools (compilers, libraries, editors, etc.)?



Which programming languages and parallelization paradigms are you using primarily?



*CSCS – Swiss Supercomputing Centre

C++ in a nutshell

"If programming in Pascal is like being put in a straightjacket, then programming in C is like playing with knives, and programming in C++ is like juggling chainsaws."

Anonymous.

“Hello World” program

```
// my first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
  return 0;
}
```

A c++ file has to be stored with the ending *.cpp,

e.g. helloworld.cpp

We will examine this program in a little later.

How to compile and execute a program

Compile = translate a program from a programming language to machine language.

CPP Compilers:

g++, icpc, ...

How do we compile e.g. helloworld.cpp

```
compiler [options] program-name [-o executable-name]
```

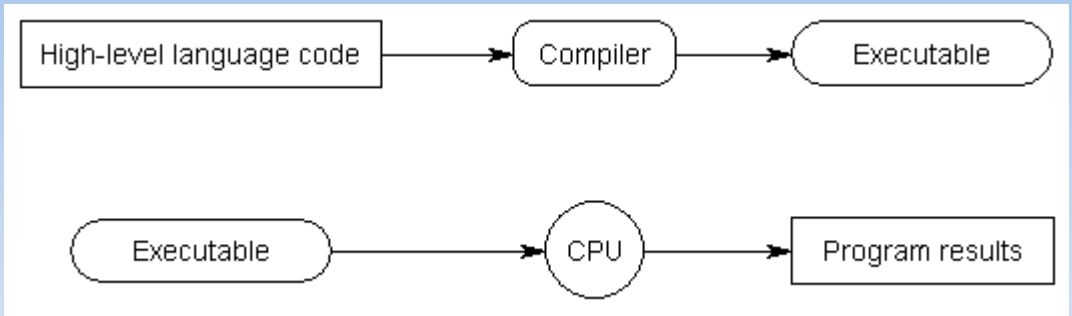
concretely:

```
g++ helloworld.cpp -o helloworld.x
```

The compiled program can then be executed by:

```
./helloworld.x
```

The compiling process

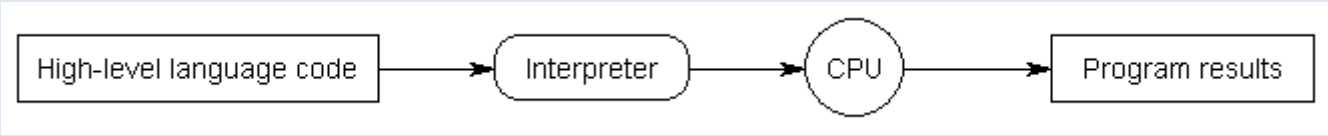


A computer's CPU is incapable of speaking C++ (or other high-level languages).

The very limited set of instructions that a CPU natively understands is called machine code (or machine language or an instruction set).

Programs written in high level languages must be translated into a form that the CPU can understand before they can be executed. There are two primary ways this is done: **compiling and interpreting**.

Modern compilers do an excellent job of converting high-level languages into fast executables.



An interpreter is a program that directly executes your code without compiling it into machine code first. Interpreters tend to be more flexible, but are less efficient when running programs because the interpreting process needs to be done every time the program is run.

Compiling and linking

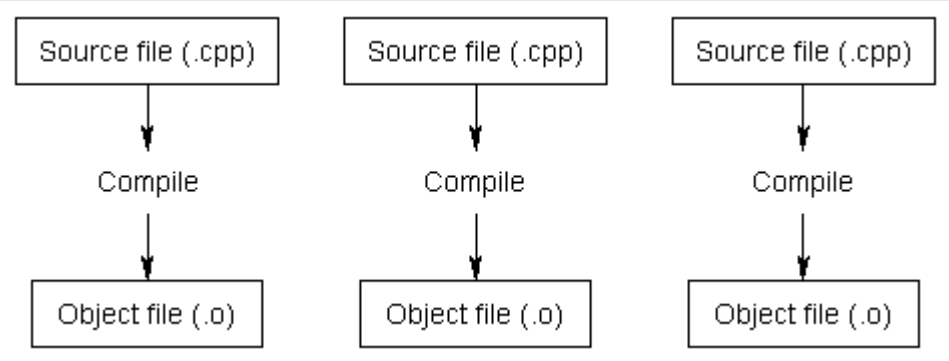
The job of the compiler is twofold:

1) To check your program and make sure it follows the rules of the C++ language.

If it does not, the compiler will give you an error to help pinpoint what needs fixing.

2) To convert each file of source code into a machine language file called an object file.

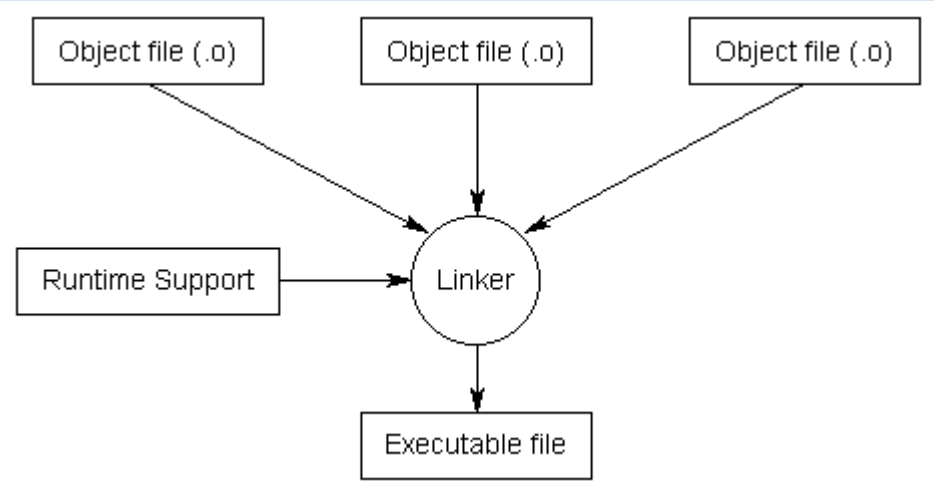
If your program had 3 .cpp files, the compiler would generate 3 object files.



Compiling and linking (2)

For complex projects, some development environments use a **makefile**, which is a file that tells the compiler which files to compile (see examples later on).

Linking is the process of taking all the object files generated by the compiler and combining them into a single executable program that you can run. This is done by a program called the linker.



Once the linker is finished linking all the object files (assuming all goes well), you will have an executable file.

The compile and link steps can be combined together if desired

Compiling Code with a makefile

In case your program consists of many routines (files), compiling by hand gets very cumbersome

```
> g++  -o abc abc.cpp a.cpp b.cpp c.cpp
```

→ **A makefile is just a set of rules to determine which pieces of a large program need to be recompiled, and issues commands to recompile them**

→ For large programs, it's usually convenient to keep each program unit in a separate file. Keeping all program units in a single file is impractical because a change to a single subroutine requires recompilation of the entire program, which can be time consuming.

→ When changes are made to some of the source files, only the updated files need to be recompiled, although all relevant files must be linked to create the new executable.

Compiling Code with a makefile (2)

Basic makefile structure: a list of rules with the following format:

target ... : prerequisites ...
<TAB> construction-commands

A **"target"** is usually the name of a file that is generated by the program (e.g, executable or object files). It can also be the name of an action to carry out, like "clean".

A "prerequisite" is a file that is used as input to create the target.

```
# makefile : makes the ABC program

abc : a.o b.o c.o  ### by typing „make“, the makefile generates an executable denotes as „abc“

g++ -o abc a.o b.o c.o

a.o : a.cpp

        g++ -c a.cpp

b.o : b.cpp

        g++ -c b.cpp

c.o : c.cpp

        g++ -c c.cpp

clean :   ### by typing „make clean“, the executable, the *.mod as well as the *.o files are deleted

        rm *.mod *.o abc
```

Compiling Code with a makefile (3)

- By default, the first target listed in the file (the executable abc) is the one that will be created when the make command is issued.
- Since **abc** depends on the files **a.o, b.o** and **c.o**, all of the .o files must exist and be up-to-date. make will take care of checking for them and recreating them if necessary. Let's give it a try!
- Makefiles can include **comments** delimited by hash marks (#).
- A **backslash (\)** can be used at the end of the line to continue a command to the **next physical** line.
- The make utility **compares** the modification time of the target file with the modification times of the prerequisite files.
- Any prerequisite file that has a more recent modification time than its target file forces the target file to be recreated.

→ A lot more can be done with makefiles (beyond the scope of this lecture)

“Hello World” revisited (2)

```
// my first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
}
```

Line 3: A blank line.

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: int main ()

This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a **type (int), a name (main) and a pair of parentheses (())**, optionally including parameters.

The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code..

“Hello World” revisited (3)

```
// my first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
  return 0;
}
```

Lines 5 and 8: { and }

The open brace ({) at line 5 indicates the beginning of main's function definition, and the closing brace (}) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when main is called. All functions use braces to indicate the beginning and end of their definitions.

Line 6: std::cout << "Hello World!";

This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

Statements in C++ are terminated by a semicolon.(in contrast to FORTRAN)

“Hello World” revisited (1)

```
// my first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";
}
```

Line 1: // my first program in C++

Two slash signs indicate that the rest of the line is a comment

Line 2: #include <iostream>

Lines beginning with a **hash sign (#) are directives read and interpreted by what is known as the preprocessor**. They are special lines interpreted before the compilation of the program itself begins.

In this case, the directive #include <iostream>, instructs the preprocessor to include a section of standard C++ code, known as header iostream, that allows to perform standard input and output operations, such as writing the output of this program (Hello World) to the screen.

C++ “Alphabet“/Operators

Valid characters in CPP (**CASE SENSITIVE!**)

0-9, a-z, A-Z

+ - * / ( ) .{} = , ' ! ? “ $: % & ; > <

Arithmetic operators:

(+, -, *, /, %)

Relational operators:

== != < <= > >=

Logical/Boolean operators:

&&, ||, !

Data types

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Declaration of variables

A variable in C++ is a name for a piece of memory that can be used to store information.

In order to define a variable, we generally use a declaration statement. Here's an example of defining variable **x as an integer variable** (one that can hold integer values): **int x;**

One of the most common operations done with variables is assignment. To do this, we use the assignment operator, more commonly known as the = symbol. For example: **x = 5;**

When the CPU executes this statement, it translates this to "put the value of 5 in memory location xyz.

Later in our program, we could print that value to the screen **std::cout << x**; // prints the value of x to the console

```
int y;  // define y as an integer variable
y = 4; // 4 evaluates to 4, which is then assigned to y
y = 2 + 5;  // 2 + 5 evaluates to 7, assigned to y
int x; // define x as an integer variable
x = y; // y evaluates to 7 (from before), assigned to x.
x = x; // x evaluates to 7, assigned to x (useless!)
x = x + 1;// x + 1 evaluates to 8, assigned to x.
```

Uninitialized variables can lead to interesting (and by interesting, we mean unexpected) results.

cout, cin, endl, the std namespace

```
#include <iostream>
int main()
{
    using namespace std;
    cout << "Hello user!"; //no std:: prefix is needed!
    cout << "Enter a number: "; // ask user for a number
    int x = 0;
    cin >> x; // read number from console and store it in x
    cout << "You entered " << x << endl;
    return 0;
}
```

If – branching & functions

```
#include <iostream>
// function that returns true if x and y are equal, false otherwise
bool isEqual(int x, int y)
{
    return (x == y); // operator== returns true if x equals y, and false otherwise
}
int main()
{
    using namespace std;
    cout << "Enter an integer: ";
    int x;
    cin >> x;

    cout << "Enter another integer: ";
    int y;
    cin >> y;

    bool equal = isEqual(x, y);
    if (equal)
        cout << x << " and " << y << " are equal" << endl;
    else
        cout << x << " and " << y << " are not equal" << endl;
    return 0;
}
```

Loops

Loop construction:

```
for ( init; condition; increment )
{
   statement(s);
}
```

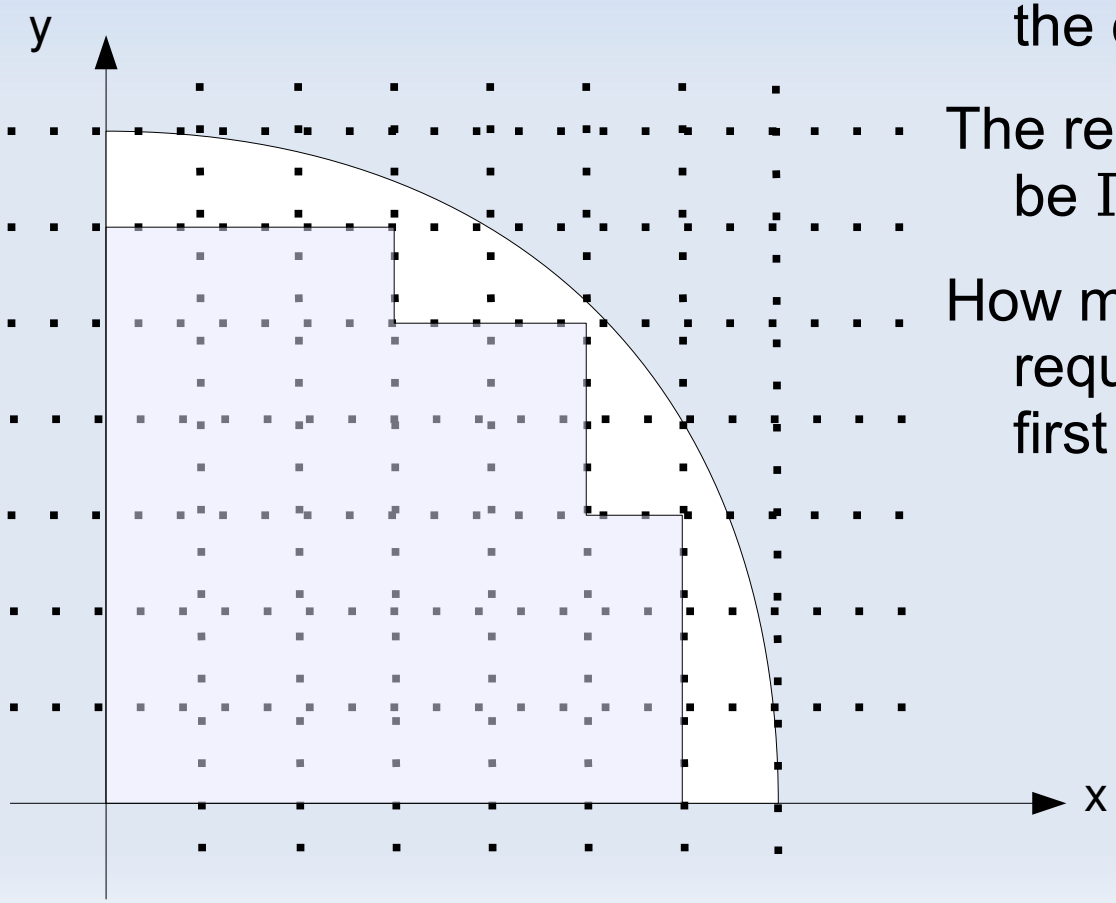
Example

```
#include <iostream>
using namespace std;

int main ()
{
   // for loop execution
   for( int a = 10; a < 20; a++)
   {
       cout << "value of a: " << a << endl;
   }
   return 0;
}
```

Exercise 1: Approximate Π

Write a program that computes the area of the unit circle by Monte Carlo integration (example code: lectures/parallel_1/Exercise_day1/supplementary_material/pi_rand.cpp).



simplification: compute a quarter of the circle.

The result should be approximately be $\Pi/4$.

How many sample points **n** are required in order to obtain the first two correct digits of Π?

Exercises 2 & 3

Exercise 2: Write a program that reads arbitrary values a, b, c and prints the solution to the quadratic Equation.

$$ax^2+bx+c=0$$

Exercise 3: Find the zero of the function $\cos(x)-x$ using Newton's Method.